Processamento de áudio em tempo real em sistemas Android

André Jucovsky Bianchi[1]

[1]Instituto de Matemática e Estatística da Universidade de São Paulo

ajb@ime.usp.br

Abstract. *Applications developed for Android systems can access audio signals that come from the microphone or from stored on files in the memory, and also reproduce signals through speakers or earphones. This article shows how it is possible to use Android's API to obtain, manipulate and reproduce signals in real time, and also points out other possible approaches for the implementation of a real time signal processing infrastructure.*

Resumo. *Aplicações desenvolvidas para sistemas Android podem acessar sinais de áudio provenientes do microfone ou de arquivos gravados na memória e também reproduzir sinais através das caixas de som ou saída de fone de ouvido. Este artigo mostra como é possível utilizar a API do Android para obter, manipular e reproduzir sinais em tempo real e aponta ainda outras abordagens possíveis para a implementação de uma infraestrutura de processamento de sinais em tempo real.*

1. Introdução

Com o aumento do uso de sistemas Android e a diversidade de modelos de aparelho, esta plataforma se torna de especial interesse para o desenvolvimento de aplicações musicais em tempo real. A capacidade de obtenção e emissão de sinais de áudio, a grande oferta de sensores e a portabilidade são características que possibilitam o uso de sistemas Android nos mais diversos cenários.

Para ampliar a utilização de uma aplicação, é interessante que ela seja desenvolvida de forma a poder ser executada em instalações padrão dos sistemas Android (ou seja, sem o uso de chamadas de sistema que necessitem de privilégios especiais). Por outro lado, é possível melhorar o desempenho de uma aplicação fazendo-a rodar com mais privilégios e utilizando funcionalidades do sistema que em geral não ficam disponíveis para aplicações comuns, como veremos mais adiante.

Este artigo se encaixa no contexto de uma pesquisa mais ampla sobre desempenho de sistemas computacionais para processamento de sinais em tempo real. Algumas iniciativas têm adaptado sistemas tradicionais de processamento em tempo real para a plataforma Android, como por exemplo Pure Data[Brinkmann 2012] e Csound[YI and LAZZARINI 2012]. Apesar disso, pouco pode ser encontrado sobre a realização de processamento em tempo real utilizando somente a API do Android, sem a utilização de outras bibliotecas ou de maiores privilégios no sistema. A documentação encontrada foi principalmente aquela disponibilizada no website oficial do sistema Android[1] e em fóruns na internet.

[1]http://developer.android.com/reference/packages.html

Apesar do enfoque geral da nossa pesquisa tratar de avaliação de desempenho, neste artigo não serão apresentados resultados numéricos, pois este texto tem como objetivo contribuir com a disponibilização de informação organizada sobre processamento de sinais em tempo real na plataforma Android. Nas próximas seções descreveremos algumas classes da API do Android com as quais se pode rapidamente construir uma pequena infraestrutura para processamento de sinais em tempo real. Falaremos sobre formas de obtenção de amostras de áudio (seção 2), agendamento de funções para manipulação destas amostras (seção 3) e reprodução do resultado (seção 4).

Os trechos de código que serão apresentado ao longo do texto representam apenas os blocos fundamentais para o processamento em tempo real utilizando a API Java do Android, e devem ser complementados para gerar uma aplicação funcional. O código completo com a aplicação que desenvolvemos para avalização de desempenho está publicado sob licenças livres[2], de forma que pode ser copiado, modificado e distribuído.

2. Fontes de sinais de áudio

O nível de abstração proporcionado pela utilização da API em Java em um sistema operacional baseado em Linux (e portanto nos padrões POSIX[3]) permite o acesso a diversas fontes de sinais de áudio. Algumas limitações são impostas pelo esquema de permissões do sistema mas, como veremos, não impedem a implementação de alternativas para acesso às diversas fontes de sinais.

Possíveis fontes de sinais de áudio são arquivos de áudio (compactados ou não), microfones e sinais transmitidos por rede (wi-fi, 3G, etc). Cada tipo de fonte será descrita separadamente nas próximas seções.

2.1. Obtendo sinais de áudio do microfone

O acesso ao sinal de áudio capturado pelo microfone é feito através da classe `AudioRecord`[4], que permite a configuração de diversos parâmetros para acesso a um buffer com os valores das amostras. Na instanciação do objeto, devem ser informados a forma de acesso ao microfone (detalhada a seguir), a taxa de amostragem desejada, o número de canais, o formato das amostras de áudio (8 ou 16 bits) e o tamanho do buffer onde serão escritas as amostras.

O microfone de um aparelho móvel é geralmente utilizado para capturar a voz do usuário em diferentes situações como, por exemplo, durante uma chamada, durante a gravação de lembretes ou para processamento por aplicativos de reconhecimento de voz. É por causa dessa diversidade de aplicações que é necessário informar ao construtor da classe `AudioRecord` a forma de acesso ao(s) microfone(s). As opções possíveis são valores constantes da classe `MediaRecorder.AudioSource`[5], estão sujeitas a disponibilidade em cada modelo de aparelho, e são as seguintes:

- `MIC`: Áudio bruto do microfone.

[2]http://www.ime.usp.br/ ajb/repo/DspBenchmarking.git/
[3]http://standards.ieee.org/develop/wg/POSIX.html
[4]https://developer.android.com/reference/android/media/AudioRecord.html
[5]https://developer.android.com/reference/android/media/MediaRecorder.AudioSource.html

- `CAMCORDER`: Áudio bruto capturado de um microfone com a mesma orientação da câmera (se disponível).
- `VOICE_DOWNLINK`: Áudio bruto recebido durante uma chamada.
- `VOICE_UPLINK`: Áudio bruto enviado durante uma chamada.
- `VOICE_CALL`: Soma dos sinais de áudio brutos enviado e recebido durante uma chamada.
- `VOICE_COMMUNICATION`: Áudio do microfone pré-processado para comunicação por voz (cancelamento de eco, controle de ganho automático, etc).
- `VOICE_RECOGNITION`: Áudio do microfone com pré-processado para reconhecimento de voz.

Um exemplo de como pode ser feita a instanciação de um objeto da classe `AudioRecord` pode ser visto a seguir:

```
AudioRecord recorder = new AudioRecord(
  AudioSource.MIC,                  // Fonte de audio.
  44100,                            // Taxa de amostragem (Hz).
  AudioFormat.CHANNEL_IN_MONO,      // Configuracao de canais.
  AudioFormat.ENCODING_PCM_16BIT,   // Codificacao dos canais.
  AudioRecord.getMinBufferSize(     // Tamanho do buffer de audio.
    44100,
    AudioFormat.CHANNEL_IN_MONO,
    AudioFormat.ENCODING_PCM_16BIT));
```

Instanciação de `AudioRecord`

Vale comentar que configurações de gravação com taxa de amostragem igual a 44.100 Hz, número de canais igual a 1 e codificação em PCM 16 bits são as únicas que têm garantia de funcionamento em todos os dispositivos. Note também que existe um tamanho mínimo para o buffer de áudio (utilizado internamente pelo gravador), que pode ser obtido através do método estático `AudioRecord.getMinBufferSize()`. Apesar da existência de um tamanho mínimo para este buffer interno de áudio, é importante saber que não há garantia de gravação sem perda de amostras se o sistema estiver com muita carga.

A leitura do valor das amostras é feita através do método de instância `AudioRecord.read()`, que recebe como argumentos um buffer para onde devem ser copiadas as amostras, o valor do índice a partir do qual os dados devem ser escritos no buffer e a quantidade de amostras a serem lidas do microfone. É importante lembrar que as amostras são codificadas como áudio PCM, e portanto são representadas por `bytes` (com valores entre -128 e 127) ou `shorts` (com valores entre -32768 e 32767), e portanto algum cuidado tem que ser tomado quando do processamento destas amostras para que não haja problemas com os limites de cada tipo de representação.

Um exemplo de método para leitura dos valores de amostras do microfone e escrita em um buffer para posterior processamento pode ser visto abaixo:

```
public void readLoop(short[] inputBuffer, int readBlockSize) {
  while (isRunning) {
    recorder.read(
```

```
4         inputBuffer, // o buffer de amostras brutas do microfone.
5         ((j++) % buffer.length) * readBlockSize, // indice do buffer.
6         readBlockSize); // numero de amostras a serem lidas.
7     }
8 }
```

Note que seguidas leituras de blocos de tamanho `readBlockSize` serão feitas, até que uma variável de controle seja tornada falsa para interromper a leitura. O valor de `readBlockSize` será discutido na seção 3, assim como a questão da conversão de representação PCM para ponto flutuante e vice-versa.

2.1.1. Gravando o sinal do microfone em um arquivo

Existe uma opção de mais alto nível para capturar, codificar e armazenar o sinal do microfone, sem no entanto permitir o acesso ao valor das amostras em tempo real. A classe `MediaRecorder`[6], disponível na API do Android, pode ser utilizada para gravar o sinal do microfone em um arquivo codificado em AMR (otimizado para codificação de fala), MPEG4 ou 3GPP (utilizado em plataformas com acesso a redes 3G). Estes arquivos podem ser posteriormente abertos para leitura e processamento.

A desvantagem desta abordagem é que todos os formatos são proprietários, o que limita suas possibilidades de uso, e utilizam modelos de compactação com perdas. A alternativa para armazenar o sinal de áudio bruto é fazer a leitura da entrada com o `AudioRecord`, da forma descrita anteriormente, e implementar algum formato com compactação sem perdas, como por exemplo FLAC ou WAV.

2.2. Obtendo sinais de áudio a partir de arquivos

A API do Android não oferece suporte à leitura e decodificação de arquivos de áudio ou vídeo para acesso e manipulação do valor das amostras do sinal. Uma opção de manipulação, utilizando código nativo, será comentada na seção 5, mas exige a utilização de privilégios administrativos no aparelho. A alternativa que resta aos desenvolvedores de aplicações é implementar bibliotecas específicas para os tipos de arquivo que utilizam.

Para o contexto deste trabalho, desenvolvemos uma classe chamada `AudioStream`, na qual encapsulamos todo o acesso a sinais de áudio. O acesso ao sinal de áudio do microfone é feito através de uma subclasse chamada `MicStream`, e o acesso a sinais armazenados em arquivos WAV, por sua vez, é feito através de uma outra subclasse chamada `WavStream`.

Uma vez que todo o acesso a um sinal de áudio está encapsulado, é possível responsabilizar a classe `AudioStream` também pela manipulação e reprodução do sinal de áudio. Veremos como fazer isto nas próximas seções.

[6] https://developer.android.com/reference/android/media/MediaRecorder.html

3. Agendamento dos ciclos de processamento

Para implementar um ambiente de processamento de sinais é necessário representar no sistema algumas características do sinal como taxa de amostragem, número de canais e tipo de codificação das amostras. Além disso, o próprio processamento também possui parâmetros tais como tamanho do bloco de processamento e fator de sobreposição (dos quais falaremos mais a seguir). Com estes parâmetros, é possível determinar o período do ciclo de processamento e agendar uma função de manipulação para execução em intervalos periódicos, acessando diferentes seções de um buffer circular.

Uma função de manipulação agendada pode não ser necessária em um contexto no qual a leitura das amostras é feita em tempo real (como é o caso da leitura feita a partir do microfone), pois as amostras são entregues para processamento exatamente com a mesma taxa que devem ser devolvidas para reprodução (supondo que a taxa de amostragem de entrada e saída é a mesma). Apesar disso, quando se deseja realizar processamento em tempo real de um sinal armazenado em arquivo, todas as amostras estão disponíveis desde o começo, e portanto é necessário que o controle do período do ciclo de processamento seja independente da disponibilidade de amostras na fonte do sinal. Se este cuidado não for tomado, a alteração nos parâmetros do processamento num certo instante pode afetar amostras que correspondem a pontos ainda muito distantes no futuro.

Desta forma, nossa opção foi por agendar uma função de manipulação que acessa um buffer circular que contém o sinal a ser modificado. Um buffer circular é um vetor de tamanho finito que tem este nome por causa da forma como o acesso aos seus índices é feito. As amostras digitais de áudio geradas pelo sistema em intervalos de tempo igualmente espaçados são gravadas sequencialmente no buffer, e o incremento dos índices de leitura e escrita é feito de forma modular (utilizando como módulo o tamanho do buffer).

Cabe comentar que, segundo a documentação do Android, não existe garantia sobre a precisão do agendamento feito através das classes disponíveis na API, de forma que pode ocorrer flutuação no período de execução dos métodos agendados dependendo da carga do sistema [7]. Infelizmente, sem utilizar recursos das camadas mais baixas do sistema (para os quais seriam necessários privilégios especiais), as classes da API são as únicas opções que existem para a realização do agendamento.

3.1. Bloco de processamento e fator de sobreposição

O **bloco de processamento** é a seção do buffer circular que será considerada em cada ciclo de processamento, e seu tamanho (em número de amostras) dependendo do algoritmo que será utilizado para manipulação do sinal e do efeito pretendido. Para um tamanho de bloco de processamento de N (em amostras) e uma taxa de amostragem de R Hz, um atraso mínimo de $\frac{N}{R}$ segundos entre a entrada do sinal original e a saída do sinal modificado é inevitável, pois este é o tempo mínimo necessário para acúmulo de N amostras.

O **fator de sobreposição** é a relação entre o tamanho do bloco de processamento e o incremento no índice de leitura do buffer circular a cada ciclo de proces-

[7] https://developer.android.com/reference/java/util/Timer.html

samento. Existem diversas razões para permitir um incremento no índice de leitura diferente do tamanho do bloco de processamento. Para processamentos no domínio da frequência, por exemplo, a ideia de usar um fator de sobreposição diferente de 1 é obter uma maior resolução temporal de análise do sinal sem perder resolução espectral. Por outro lado, existem processamentos puramente temporais que também podem se beneficiar da sobreposição de blocos de processamento, como por exemplo *Time Stretching* ("esticamento" temporal) e *Pitch Shifting* (alteração de altura musical) executados no domínio do tempo utilizando *overlap-add* [Zölzer et al. 2002].

Assim, para um fator de sobreposição igual a M, o $j-$ésimo ciclo de processamento altera a seção do buffer correspondente às amostras do intervalo $\left[j\frac{N}{M}, j\frac{N}{M} + N - 1\right]$. Isto também determina o período do ciclo de processamento, que é dado por $T = \frac{N}{MR}$.

É necessário garantir que as amostras de áudio correspondentes ao $j-$ésimo ciclo de processamento já tenham sido capturadas antes do início do ciclo. Para isto, a leitura da entrada deve ser feita em blocos de tamanho N/M e deve-se aguardar que o buffer possua pelo menos N amostras antes de que o primeiro ciclo de processamento seja iniciado.

Desta forma, uma vez determinados a taxa de amostragem, o tamanho do bloco de processamento e o fator de sobreposição, pode-se agendar uma função de processamento que deve será executada, idealmente, nos valores de tempo $t_j = \frac{N}{R} + j(\frac{N}{MR})$ segundos para $j \in \mathbb{N}$. A saída destas funções deve estar disponível para reprodução imediata antes do início do próximo ciclo, e portanto o tempo efetivamente disponível para a manipulação do sinal é menor do que período do ciclo de processamento.

O agendamento pode ser feito utilizando-se a classe `ScheduledExecutorService`[8] da API do Android, que permite a indicação do período do agendamento em nanossegundos e assim possibilita um agendamento mais preciso. No trecho de código abaixo, `scheduler` é uma variável do tipo `ScheduleExecutorService` e é agendada de acordo com o cenário que apresentamos nos últimos parágrafos:

```
1  // agenda a funcao de processamento periodica
2  public void scheduleDspCallback() {
3    System.gc(); // recolhimento de lixo
4    try {
5      scheduler = Executors.newScheduledThreadPool(1);
6      dspTask = scheduler.scheduleAtFixedRate(
7        dspCallback,                  // a funcao de processamento.
8        (float) N/R*Math.pow(10,9),   // atraso p/ primeira execucao.
9        (float) N/(MR)*Math.pow(10,9),// intervalo entre execucoes.
10       TimeUnit.NANOSECONDS);        // unidade de tempo.
11   } catch (Exception e) {
12     e.printStackTrace();
13   }
14 }
```

Agendamento da função de processamento

[8] https://developer.android.com/reference/java/util/concurrent/ScheduledExecutorService.html

A função de processamento agendada corresponde, na verdade, ao método `run()` de um objeto do tipo `Runnable`[9], que veremos com mais cuidado na próxima seção.

4. Manipulação e reprodução do sinal

A classe `AudioTrack`[10] permite a escrita de valores de amostras (codificadas em PCM 8 bits ou 16 bits) em um buffer para reprodução pelo hardware de áudio. Esta classe possui dois modos de operação, *static* e *streaming*, que diferem quanto à quantidade de áudio que será reproduzida e as condições de transferência do sinal da camada de aplicação para o hardware do aparelho. Apesar do modo *static* oferecer menor latência, para poder escrever dados para reprodução de forma contínua é necessária a utilização do modo *streaming*. A escrita das amostras para reprodução é feita através do método `write()`, que veremos em instantes. Antes, vejamos um exemplo de instanciação de `AudioTrack`:

```
AudioTrack track = new AudioTrack(
  AudioManager.STREAM_MUSIC,       // stream de audio a ser usado.
  44100,                           // taxa de amostragem.
  AudioFormat.CHANNEL_OUT_MONO,    // numero de canais.
  AudioFormat.ENCODING_PCM_16BIT, // codificacao das amostras.
  audioStream.getMinBufferSize(), // tamanho do buffer.
  AudioTrack.MODE_STATIC);         // modo de operacao
```

Instanciação de `AudioTrack`

Neste ponto, já sabemos como ler amostras de um sinal de áudio com `AudioRecord`, agendar a execução de uma função de processamento com `ScheduleExecutorService`, e preparar o sistema para reprodução instanciando `AudioTrack`. O que falta agora para completar o desenvolvimento de uma infraestrutura para processamento de sinais em tempo real é apenas a manipulação de fato do sinal e a escrita das amostras resultantes para reprodução.

Na seção 3, demos um exemplo de como agendar uma função de processamento utilizando um objeto do tipo `Runnable` chamado `dspCallback`. A seguir, mostramos uma possível implementação para este objeto, usando um buffer de entrada e outro de saída para permitir a sobreposição de blocos durante o processamento:

```
short inputBuffer;  // buffer circular de amostras de entrada
short outputBuffer; // buffer circular de amostras processadas
int j;              // indice de leitura dos buffers circulares
int L = inputBuffer.length;
Runnable dspCallback = new Runnable() {
  public void run() {
    double performBuffer[] = new double[L];
    // converte de PCM shorts para doubles
    for (int i = 0; i < N; i++)
```

[9] https://developer.android.com/reference/java/lang/Runnable.html
[10] https://developer.android.com/reference/android/media/AudioTrack.html

```
10        performBuffer[i] = (double)
11          inputBuffer[(j*N/M + i) % L] / Short.MAX_VALUE;
12      // executa o algoritmo escolhido
13      dspAlgorithm.perform(performBuffer);
14      // converte de doubles para PCM shorts
15      for (int i = 0; i < N; i++) {
16        if (i >= N-N/M) // previne overlap-add nos ultimos indices
17          outputBuffer[(j*N/M + i) % L] = 0;
18        outputBuffer[(j*N/M + i) % L] += (short) // overlap-add
19          (performBuffer[i] * Short.MAX_VALUE);
20      }
21      // escreve as amostras para reproducao
22      track.write(outputBuffer, ((j++)*N/M) % L, N);
23    }
24  };
```

Classe que implementa o ciclo de processamento

O método `run()` do objeto acima consiste em, essencialmente, quatro passos: (1) conversão das amostras de PCM para ponto flutuante para permitir sua manipulação numérica; (2) chamada do método `perform()` do algoritmo escolhido para manipulação das amostras; (3) conversão de ponto flutuante de volta para PCM; e (4) enfileiramento das amostras para reprodução. Se o fator de sobreposição for diferente de 1, então algum cuidado tem que ser tomado com a sobreposição das amostras de saída. No exemplo acima, é feita uma soma simples dos sinais sobrepostos, mas outras técnicas podem ser implementadas para obter outros efeitos ou algoritmos mais eficientes.

5. Considerações finais

5.1. Efeitos com a classe `AudioEffect`

Na seção 4, vimos como usar `AudioTrack` para escrever amostras brutas para reprodução. Uma outra possibilidade, para reprodução de arquivos codificados, é a utilização da classe `MediaPlayer`[11], que permite um gerenciamento de alto nível da reprodução de arquivos de mídia. Tanto `AudioTrack` quanto `MediaPlayer` permitem a aplicação de alguns efeitos de áudio aos sinais reproduzidos, através do uso de subclasses de `AudioEffect`[12].

O desenvolvimento de subclasses de `AudioEffect` é uma abordagem interessante, porém necessita de código nativo (C/C++) e pode necessitar de privilégios especiais (a documentação revela que esta infraestrutura de efeitos de áudio foi pensada para inclusão de efeitos pelos fabricantes de aparelhos). A implementação descrita neste artigo, por sua vez, é toda escrita em Java e roda com privilégios padrão de uma aplicação no sistema Android. Consideramos a investigação de processamento de áudio em tempo real com código nativo um trabalho futuro, que pode trazer melhorias de desempenho no processamento. Apesar disso, o uso de código nativo não é sinônimo imediato de ganho em desempenho, pois seu uso implica maior complexidade na aplicação e há um custo na transição entre código Java e código nativo na máquina virtual. Outros cuidados devem ser tomados para evitar sobrecarga

[11] https://developer.android.com/reference/android/media/MediaPlayer.html
[12] https://developer.android.com/reference/android/media/audiofx/AudioEffect.html

no sistema, como evitar criar objetos e alocar memória, fazer uso de ferramentas de *profiling*, minimizar o uso de números de ponto flutuante, entre outros[13].

5.2. O projeto `libpd`

Desde o fim de 2010 o projeto `libpd`[14] reempacota o código fonte do Pure Data[15] e oferece toda sua funcionalidade de processamento de áudio através de uma biblioteca que pode ser acessada a partir de outras linguagens e ambientes. Um dos ramos do projeto mantém uma versão da biblioteca para Android, que poder ser utilizada para a construção de aplicações musicais [Brinkmann 2012].

Esta é uma iniciativa que utiliza as vantagens do código nativo (o Pure Data é escrito em C) e conta com um modelo de implementação de processamento de sinais em tempo real robusto e amplamente utilizado, além de permitir que *patches* do Pure Data sejam executados em ambientes Android. Um estudo mais aprofundado sobre as possibilidades e o desempenho que podem ser obtidos utilizando a `libpd` seria de ótima valia para a ampliação do uso de sistemas móveis como processadores de sinais digitais em tempo real.

Referências

Brinkmann, P. (2012). *Making Musical Apps*. O'Reilly Media.

YI, S. and LAZZARINI, V. (2012). Csound for android. *Proceedings of the Linux Audio Conference 2012*.

Zölzer, U., Amatriain, X., Arfib, D., Bonada, J., De Poli, G., Dutilleux, P., Evangelista, G., Keiler, F., Loscos, A., Rocchesso, D., Sandler, M., Serra, X., and Todoroff, T. (2002). *DAFX:Digital Audio Effects*. John Wiley & Sons.

[13] https://developer.android.com/guide/practices/design/performance.html
[14] https://github.com/libpd
[15] http://puredata.info